Contents

Introduction 2

How to include POD in your program 3

Running POD 4

Accessing Pascal statements 4

Accessing Pascal variables 5

POD commands
- B - Set/Clear Breakpoints 7
- C - Continue 9
- D - Display Parameters 9
- G - Go or Go to a Label 10
- H - Execution History 11
- K - Kill Breakpoints and Labels 12
- L - Label Statement 12
- P - Single Procedure Step 13
- R - Register Dump 13
- S - Single Step 14
- T - Trace Mode 14
- V - Variable Watch 15
- W - Write Variables 17

Advanced debugging techniques 18

OMSI Pascal on-line debugging system documentation.
Copyright 1979 Oregon Software.
OMSI Pascal is a trademark of Oregon Software.

Introduction

The Pascal On-line Debugging system (POD) is a symbolic debugging tool that lets you interactively control the execution of your Pascal program. You can suspend execution at particular statements, execute one statement at a time, and examine and modify the values of particular variables. Since POD traps errors and identifies the last statement executed, you can easily pinpoint the source of run-time errors.

POD is really a series of Pascal procedures which are linked with a program. When you specify the debugging option (/D), the Pascal compiler includes a call to POD before each procedure and statement in your program. This lets POD control program execution. The compiler also produces a symbol table file containing the definitions and locations of all variables and procedures in your program. Using this, POD can find and modify variables and refer to procedures by name.

How to include POD in your program

To use POD, you must compile your program with the debugging switch, /D. This automatically generates a symbol table file for your program. For example:

```
.R PCL
*PASCAL/D TRIM/D
```

The /D switch causes debugging instructions to be included in the compiled program. The /D switch also produces a debugging file (TRIM.SYM) containing the symbol table information for the procedures and variables of TRIM.

POD supports an option called source debugging, selected using the /S compilation switch. This lets POD print the Pascal source lines associated with the compiled statements in your program. With the /S switch you can debug a program without having to print a listing of the program. The cost for using source debugging is an increase in the size of the program being debugged and a somewhat slower execution speed. All of the examples in this manual use the source debugging option.

If you wish to use the source debugging option, specify both the /S and /D switches in the compilation command:

```
.R PCL
*PASCAL/S/D TRIM/S/D
```

When the /S source debugging option is selected, a listing file (TRIM.LST) is automatically created. POD reads this file to display the source program for each Pascal statement. If the listing file is deleted, source debugging is automatically disabled, and POD will then identify statements only by procedure name and statement number.

Running POD

When your program starts executing, POD will identify itself and ask you for the name of your program. It is assumed that the symbol file and listing file (if the S option is invoked) will share the program name. If either file cannot be found, POD will ask specifically for the necessary file name. If POD asks for a listing file and none exists, give a carriage return. This will cancel the source debugging option. POD will then ask for a symbol file name. Here is a typical POD opening dialogue:

```
RUN TRIM
POD (Pascal On-line Debugger) - 24-Apr-79
POD - program name? TRIM
}
```

When POD is ready to accept commands, it will prompt you with a right brace (}). On some terminals this will print as a right square bracket (]). Commands to POD may be typed in either lower or upper case, and spaces in the commands are ignored. Several POD commands can be typed on the same line by separating the commands with semi-colons (;).

POD commands are presented alphabetically beginning on page 7.

Accessing Pascal statements

POD identifies Pascal statements by the name of the procedure containing the statement and the number of the statement in the procedure. The statement number can be found in the column labeled STMT in the listing file produced by the Pascal compiler. Statements in the main body of a Pascal program are considered to be in the procedure MAIN. All Pascal programs begin executing at MAIN,1.

If the source debugging option is being used, POD will usually print the the source line along with the procedure name and statement number.

Pascal allows you to define procedures which define other local procedures. In this way it is possible to create a program containing several procedures all having the same name. It is strongly recommended that all of the procedures in your program have unique names in order to avoid confusion during debugging.

Accessing Pascal variables

POD lets you access the variables in your program in much the same way as you use variables in Pascal. Variables and procedure parameters are identified by name, such as MARGIN, LIMIT, or SHOESIZE. Records are specified using the standard dot notation such as: COORD.X, and RANGE.TOLERANCE.LOW. POD will generate an error message if too few (or too many) fields are specified for a record. Arrays of multiple dimensions are allowed, and POD will check the data type and limits of each index when accessing arrays. Pointers are specified in the usual way. The value of the pointer itself is interpreted as a decimal integer. A nil pointer has a value of zero, and POD will generate an error message if a reference through a nil pointer is attempted.

You can access very complex structures by combining several of the structures described above. In general, POD can access a variable in a structure in the same way as that variable is used in your program. Examples of legal variables are shown below:

```
FEET
A.B.C.D
CHIP↑.TEMPLATE[3,1,-5].FLUX
PTR↑.SON↑.SON↑.SON
```

Integers are treated as 16 bit signed numbers. Octal integers are specified by placing a "B" after the integer such as 377B. Boolean variables take values of either TRUE or FALSE. Character data, including character strings, are always enclosed within single quotes as with 'X' and 'THIS IS A TEST'. Spaces are not ignored within a character string. Real variables are used in the usual way. POD can also access scalar types defined by the user. For example, consider the program section below:

```
TYPE
    COLOR=(RED, WHITE, BLUE);
VAR
    X: COLOR;
```

When POD displays the value of X, it will correctly print the scalar type of X. This capability is provided only by POD -- Standard Pascal does not permit output of scalar types.

```
} X:=RED
} W(X)
RED
}
```

POD has another facility not available to the Pascal programmer: its ability to display the value of sets. The ".." notation for included set elements is available for both the input and output of set values.

```
TYPE
    COLOR=(RED, ORANGE, YELLOW, GREEN, BLUE);

VAR
    RB: SET OF COLOR;
    VALUES: SET OF INTEGER;
    Q: SET OF CHAR;
```

These variables may be accessed by POD as shown below:

```
) RB:=[RED..YELLOW,BLUE]
) W(RB)
[RED..YELLOW,BLUE]
) VALUES:=[1..20,50,40,30]; W(VALUES)
[1..20,30,40,50]
) Q:=['E','A','C','F','B','D']
) W(Q)
['A'..'F']
)
```

As demonstrated above, POD lets you assign values to variables in the same way as you assign values to variables in your program. The only restriction is that you cannot evaluate expressions such as C:=A+B, and you cannot call functions such as R:=SIN(3.1415).

POD enforces the Pascal scope rules. In general, this means that at any point in your program you can only access the variables that the program itself can access at that point. Global level variables, those defined at the start of the program, are always available. However, as different procedures are executed, the local variables and arguments of those procedures are temporarily available, while the local variables in procedures not being executed are never available. If you try to use a variable which is not available, POD will print a "symbol not found" error message. Remember, at any statement, you can only use the variables that are available to the program at that point.

POD lets you directly address memory locations as integers. For example, 1234B:=240B modifies location 1234 (octal) to contain 240 (octal). This feature is most commonly used when dealing with pointers. However, be careful, for you might accidently modify a location within your program and cause unpredictable results.

B(): Set/Clear Breakpoints

The "B" command sets a breakpoint at a particular statement within a program. Before POD executes each statement in your program it checks to see if a breakpoint has been set at that statement. If a breakpoint has been set, POD suspends the execution of the program and enters command mode. At this point you can examine and alter variables, check the history of the program's execution, or continue the execution of the program.

To set a breakpoint at a statement, type a "B" followed by the statement identifier (procedure and statement number) contained within parentheses. POD will interrupt the execution of your program just before the statement at which a breakpoint is set. Up to eight breakpoints may be in effect at any one time. Examples:

```
} B(MAIN,1)
} G
Breakpoint at MAIN,1  BEGIN I:=0;
} B(INIT,5); C
Breakpoint at INIT,5  PARAM1:=0; PARAM2:=0;
}
```

(The examples above show how the source debugging option works. When POD stops at breakpoint, it prints the Pascal source line for that statement.)

The "G" command in the example starts program execution. The "C" command continues from the breakpoint.

POD has the capability to execute a series of POD commands when a breakpoint is encountered. This facility, called stored commands, is specified by placing the command within angle brackets (< >) after the break command as shown here:

```
} B(MAIN,6) < W(DEPTH); DEPTH:=5 >
} B(POSITION,32)<W(X,Y);C>
```

The first example displays the value of the variable DEPTH then assigns the value of 5 to DEPTH each time the program comes to the statement at MAIN,6. The second example displays the values of the variables X and Y and then continues the execution of the program. In this case POD will not stop and enter command mode. Instead, each time the program comes to the statement at POSITION,32, the variables X and Y will be displayed and the program will continue.

Any POD command may appear in a stored command, but stored commands may not be nested, ie. a stored command may not define other stored commands. As many POD commands as will fit on a single line may be specified in a stored command.

There are two ways to cancel a breakpoint. The "K" command described below can be used to kill all breakpoints or just a single breakpoint. However, if the program has just been interrupted because a breakpoint was reached, that breakpoint can be cancelled by using the "B" command with no arguments.

```
} B(MAIN,1)
} G
Breakpoint at MAIN,1  BEGIN I:=0;
} B
} C
```

The "D" command may be used to display the currently active breakpoints and their associated stored commands.

C: Continue execution

If the execution of your program has been suspended by POD, you may use the "C" command to resume execution of the program. If your program has not started executing, either the "C" or the "G" command may be used to start the program. The section above describing breakpoints has several examples which use the "C" command. Once your program has terminated and POD has re-entered command mode, any attempts to continue the program with the "C" command will be ignored. (There is nowhere to go!) The program may, however, be restarted with the "G" command described below.

If you set a breakpoint inside a loop, it is sometimes desirable to let the statement at the breakpoint execute several times before stopping. One way to do this is to use the "C" command several times to continue from the breakpoint until the desired iteration in the loop is reached. Another solution is to use a repeat count contained inside parentheses after the "C". The repeat count tells how many times the statement at which the breakpoint has been set should be executed before the breakpoint takes effect. For example, you can set a breakpoint at COUNT,10 which is inside a loop structure. When the loop is first entered, POD will stop the program at COUNT,10 with a breakpoint. The command C(6) will let the loop iterate 6 times before the program stops again at COUNT,10 with a breakpoint. Each of the eight breakpoints has its own repeat count.

D: Display POD Parameters

The "D" command displays the watched variables, labels, and breakpoints which are currently active. Watched variables are described below in the section about the "V" command. Labels are discussed below in the sections about the "G" and "L" commands. The stored commands associated with breakpoints and the watched variable are also displayed.

```
) D

Watching: B[5] (W(B[6],B[7],B[8]))

Breakpoints:
MAIN,13 (W(FOO);C)
MAIN,20
ERR,5   (W(ERRORCODE);H)

User defined labels:
1:  MAIN,1   BEGIN I:=0;
5:  RETRY,3  RESET(F,NAME,'DAT',STATUS);
)
```

G, G(): Go or Go to a Label

The "G" command without arguments starts or restarts your program at MAIN,1. If the "G" command is followed by a label number in parentheses, the program will be continued at that user defined label. Do not confuse user defined labels with Pascal statement labels. User defined labels are created with the "L" command dynamically as POD controls your program. Pascal statement labels are defined in your source code and are used by the PASCAL compiler to generate targets for the Pascal GOTO command. POD does not use Pascal statement labels.

The "L" command labels the program statement about to be executed. The most common way to define a label at a particular statement is to set a breakpoint at that statement, execute the program until that statement is reached, and then use the "L" command to define the label.

The "G" command should be used with care. It is not always possible to branch from any Pascal statement to any other Pascal statement. Labels follow the same scope rules as variables, so depending on which procedures are being executed, some labels may not be available. If you try to go to a label which is not available, POD will respond with the error message "You can't get there from here". One reason that POD cannot go to a particular label is that if the label is in a procedure which is not being executed, POD is not able to invent the values of the local variables associated with that procedure.

```
) B(MAIN,5); C
Breakpoint at MAIN,5  J:=SIN(Q);
) L(3); B(MAIN,27); C
Breakpoint at MAIN,27  WRITELN('X)Y');
) G(3)
Breakpoint at MAIN,27  WRITELN('X)Y');
) G
Breakpoint at MAIN,5  J:=SIN(Q);
)
```

H: Print Program Execution History

POD maintains a list of the last 10 statements executed by a program. This history is useful in determining how the program got to a breakpoint or how it got to a statement which caused an error. The "H" command prints the history and also the procedure execution stack. The stack shows the procedure and function nesting all the way back to the main body of the program.

```
} B(EVALUATEBOARD,1);C
Breakpoint at EVALUATEBOARD,1  FOR I:=-5 TO 49 DO BMAN[I]:=FALSE;
} H
Program execution history

GENMOVE,3  BEGIN
GENMOVE,4  FATHER:=F;
GENMOVE,5  MOVE:=I*256+J;
GENMOVE,6  OLDPIECE:=B[I]; B[I]:=EMPTY;
GENMOVE,7  OLDPIECE:=B[I]; B[I]:=EMPTY;
GENMOVE,8  IF TURN=BLACK THEN
GENMOVE,9  IF J<=8 THEN B[J]:=BLACKKING ELSE B[J]:=OLDPIECE
GENMOVE,11  IF J<=8 THEN B[J]:=BLACKKING ELSE B[J]:=OLDPIECE
GENMOVE,15  VALUE:=EVALUATEBOARD(ENEMY);
EVALUATEBOARD,1  FOR I:=-5 TO 49 DO BMAN[I]:=FALSE;

Procedure execution stack

EVALUATEBOARD,1  FOR I:=-5 TO 49 DO BMAN[I]:=FALSE;
GENMOVE,15  VALUE:=EVALUATEBOARD(ENEMY);
MOVEPIECE,11  IF MOVESALLOWED THEN GENMOVE(I,J);
EXPAND,15  IF COLOR[WHO]=TURN THEN MOVEPIECE(I,I,0,0);
MAIN,7  EXPAND(ROOT,TRUE);
}
```

K, K(): Kill Breakpoints and Labels

When the "K" command is given without arguments, all label definitions and breakpoints are deleted. When the "K" command is followed by a statement identifier, the breakpoint at that statement is removed.

```
} B(MAIN,5)
} K(MAIN,5)
} B(MAIN,17)
} K
```

Individual breakpoints can also be removed with the "B" command.

L(): Label a Statement

You may label up to eight statements with the "L" command. Labels are used as targets of the "G" command. The label number (1 through 8) is placed in parentheses after the "L". The "L" command always defines the label at the current location within the program being executed. Check the description of the "G" command above for a warning about branching within a Pascal program. The "D" command may be used to list the currently active labels.

```
} B(MAIN,13); G
Breakpoint at MAIN,13  A:=1;
} L(1)
} B(MAIN,15); C
Breakpoint at MAIN,15  B:=37;
} L(5)
} D

Breakpoints:
MAIN,13
MAIN,15

User defined labels:
1:  MAIN,13  A:=1;
5:  MAIN,15  B:=37;
}
```

P, P(): Execute one Statement in Current Procedure

The "P" command executes a single statement in the current procedure. "P" will not single step through functions and procedures nested in the current procedure, but instead will treat their calls as single statements. If the current procedure ends, "P" will begin single stepping the procedure that called the current procedure. (Compare "P" to the similar "S" command described below.)

If a repeat count is given in parentheses after the "P", the specified number of statements will be executed before stopping. As with the "C" command, you may not proceed past the end of the program once the program has terminated. Use the "G" command to restart the program.

```
} P
Breakpoint at MAIN,1  BEGIN I:=0;
} P
Breakpoint at MAIN,2  J:=RANDOMINTEGER(3);
} P
Breakpoint at MAIN,3  K:=J*J-I;
} P(5)
Breakpoint at MAIN,8  IF K<J THEN BEGIN
}
```

R: Register Dump

The "R" command prints the values of the processor registers R0-PC in both octal and decimal. This command is normally useful only to those programmers who include in-line assembly language code in their Pascal programs.

S, S(): Single Step

The "S" command is identical to the "P" command above, except that if a statement being stepped through contains a procedure or function call then the new procedure or function will be executed one step at a time. As with "P", a repeat count may be specified.

```
} S
Breakpoint at MAIN,1  BEGIN I:=0;
} S
Breakpoint at MAIN,2  RANDOMINTEGER(3);
} S(1)
Breakpoint at RANDOMINTEGER,1  BEGIN RANDOM:=X;
}
```

T(): Trace Mode

"T(TRUE)" turns on statement trace mode, while "T(FALSE)" turns it off. When trace mode is on, POD will print the location of each statement before it is executed. If several PASCAL statements appear on the same line in the source file, and if those statements are each executed in sequence, then the line containing those statements will be printed only once.

```
} B(MAIN,6)
} T(TRUE)
} G
MAIN,1  BEGIN I:=0;
MAIN,2  J:=0; K:=0; L:=3.14159;
MAIN,5  WRITELN('HI THERE');
HI THERE
Breakpoint at MAIN,6  WRITELN;
}
```

V(): Variable Watch

The "V" command makes POD watch the value of a variable. Before each statement in your program is executed, POD compares the current value of the variable with the value it had when the "V" command was given. If the value has changed, POD stops your program and tells you so. If you continue your program, POD will continue watching for a change in the variable.

The "V" command is useful if your program is malfunctioning because the value of some critical variable is being destroyed somewhere. The "V" command can also be used to watch locations in low memory to detect the incorrect use of a nil pointer.

```
) V(DEPTH)
) C
Value of "DEPTH" changed at statement:
DESCEND,1  DEPTH:=DEPTH+1;
Old value: 0
New value: 1
Breakpoint at DESCEND,2  IF DEPTH>MAXDEPTH THEN
) C
Value of "DEPTH" changed at statement:
DESCEND,1  DEPTH:=DEPTH+1;
Old value: 1
New value: 2
Breakpoint at DESCEND,2  IF DEPTH>MAXDEPTH THEN
) C
Value of "DEPTH" changed at statement:
DESCEND,38  DEPTH:=DEPTH-1;
Old value: 2
New value: 1
Breakpoint at DESCEND,39  END;
)
```

Stored commands may be specified with the "V" command in the same way as with the "B" command. The "D" command will list the name of the variable being watched and the stored commands if any were given. A variable watch is terminated by using the "V" command with no arguments. POD will automatically terminate a watch on a variable when that variable is no longer available. When POD does this, it prints the message "Watch terminated -- value didn't change".

```
} B(EVALUATEBOARD,35); C
Breakpoint at EVALUATEBOARD,35  FOR I:=5 TO 39 DO
} V(BLACKSCORE)(W(WHITESCORE))
} C
Value of "BLACKSCORE" changed at statement:
EVALUATEBOARD,224  ELSE BLACKSCORE:=BLACKSCORE+MOC4;
Old value: 0
New value: 400
Breakpoint at EVALUATEBOARD,225  IF BLACKDENY(WHITEDENY THEN
0
} C
Watch terminated -- value didn't change
Breakpoint at MAIN,28  MAXLEVEL:=0;
}
```

W(): Write Variable Value

The "W" command is used to write the value of a variable, pointer, constant, or memory location. The format of the output is determined by the type of the variable being written. For example, integer variables are written as 16 bit signed decimal integers, while set variables are written using set notation. The names of the variable to be displayed are placed inside parentheses following the "W". If more than one variable is to be written then the names are separated by commas. Physical memory locations are addressed as integers (either octal or decimal). As in Pascal, integer and real values may use format control with the colon (:) notation. This is also how one examines memory locations in octal.

```
} W(TURN)
BLACK
} W(COLOR[BLACKKING],COLOR[WHITEKING])
BLACK
WHITE
} W(USERMOVES[5])
'B1'
} W(ROOT↑.SON↑.VALUE)
402
} W(54B)
-10154
} W(54B:-1)
154126B
} W(S)
['A','M','Z']
} W(R)
 3.141593E+00
} W(CH)
'A'
} W(I)
123
}
```

Advanced Debugging Techniques

If you write large Pascal programs, you might find that you are not able to use POD to help debug your program because of memory size restrictions. However, there are several things you can do to reduce the amount of memory required by POD. The easiest thing to disable source debugging. The use of the source debugging option (/S) expands your program by one word for every Pascal statement in your program. For large programs you may save more than 1K words by not using source debugging.

Another technique you can use is selective debugging. You can edit your program to turn off the generation of POD debugging information around procedures which have already been tested and debugged. To turn off debugging, place the line {$D-} before the procedure definition and {$D+} after the procedure. You will not be able to set breakpoints or examine variables in such procedures, but you will save two or three words for every statement not debugged. Be sure debugging is enabled around all variables you may wish to examine and around the main procedure.

If you program uses overlays, you can still debug your program using POD. When you compile the main body of the program, which resides in the root segment, use the debugging switch (/D) and produce a symbol table file. Compile each of the external modules in the normal way without the debugging switch. You cannot enable debugging in external procedures because you would have to produce a symbol table file for POD. The main body of your program must also have a symbol table, and there is no way to combine the two into a single usable file. The only way to debug an external procedure is to include its definition in the main program. In other words, you must make the procedure not be external.

When you link your overlaid program you will have to use two overlay regions to contain the modules of POD. These two overlay regions may, in most cases, also contain your own external procedures. There should be no conflicts because POD only lets you debug in the root segment, and as long as the two POD modules RTDBG and DBG are placed in the root, there should be no problems with the overlays.

You cannot set breakpoints within external procedures, but you can cause a break when the external procedure is called from the main program. This is done by setting a breakpoint and giving only the name of the procedure at which to break as with: B(OVER1). This type of breakpoint will stop the program before the external procedure OVER1 is executed. The only variables you will be able to examine and modify in OVER1 are those variables in the parameter list for OVER1. Note that the names of the parameters are defined by the external procedure definition of OVER1 in the main program, not by the definitions in OVER1 itself.